Verification of Radiation Transport Packages (U)

Mark G. Gray
X-TM
Los Alamos National Laboratory

Code verification for scientific simulation, ensuring that a code solves the equations it's supposed to solve, is an important part of software quality assurance. Verification is not merely testing: verification activities instead permeate the analysis, design, and implementation phases of software development. In the analysis and design phases Design by Contract™[1] formalizes coding requirements and assumptions that are the basis for white-box testing. In the implementation phase the analytic test method and levelization are the basis for grey-box unit testing. Finally, comparison with analytic solutions is the basis for black-box systems testing. In this paper I describe the application of these testing disciplines to radiation transport packages. (U)

Keywords: verification, validation, radiation transport

Introduction

I have been thinking about the problem of code verification for some time now. On the one hand talk of the formation of a verification and validation group indicates its importance to some, on the other hand lack of the formation of a verification and validation group indicates its unimportance to others. In either case I wondered of what importance verification was to me, a code physicist. Why is verification important? Where does verification fit in the software engineering process? And how do I fulfill my verification responsibilities?

My original idea was to decide what I should do by looking at the best practices for verification, and adopting those appropriate to my own work. Although I have remained true to this original idea, somewhere along the way I found answers to the verification problem that gave me much more than I had been looking for; I developed a deep appreciation of role of verification and validation as the very heart of the insight which computational physics can offer.

Why is Verification Important

My first unexpected insight was that if verification and validation were both necessary, it must be because **two** different models were being tested against reality.

To see this, consider the diagrammatic description of reality and the modeling processes given in Figure 1. Given some experimental initial state, I_p, physical processes, P, produce an observable output, O_p.

To predict O_p we create a mathematical model, M. By encoding the initial physical state, I_p, via E_{mp} into the initial model state, I_m, M can predict the model output, O_m, which can be decoded via D_{pm} into a physical prediction, O_p. If the model is valid its predictions match the actual physical result.

The mathematical model is much more than just a predictive tool. We visualize, discuss, debate,

[1]"Design by Contract" is a trademark of Interactive Software Engineering.

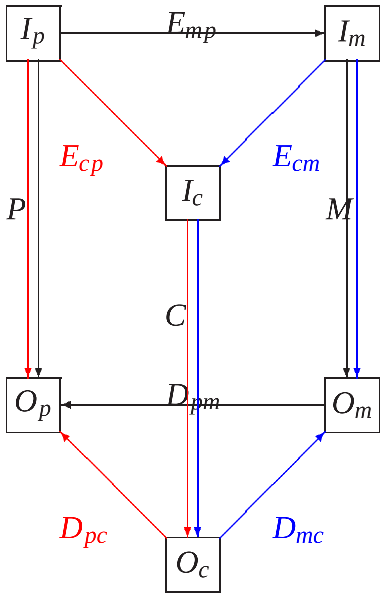

Figure 1: Structural Role of Verification and Validation. Verification ensures that the code matches the model. Validation ensures that the code matches the reality. Both are necessary for the model to provide insight.

and reason about the physical system using the mathematical model. In short, we understand the physical system through our mathematical model of it.

Unfortunately we can solve the mathematical model for only the simplest physical systems. So we construct a computational model of the mathematical model.

Verification ensures that encoding from the mathematical model to the computational model, computing, and decoding from the computation to mathematical model gives the same result as solving the (analytically unsolvable) mathematical model.

Validation ensures that encoding from the physical world to the computational model, computing, and decoding from the computational to physical world gives the same result as running the (politically inaccessible) experiment.

Verification and validation are logically necessary to ensure that the mathematical model models reality. If the model is valid, then we can use it to understand the physical system; this is the insight Hamming is talking about when he says:

The purpose of computing is insight, not numbers. –*R. W. Hamming*[Hamming, 1986]

Where does Verification Fit in the Software Engineering Process

Once I understood the structural purpose of verification and validation, my next insight was that since verification defines a standard for code quality, it should play a dynamic role in determining what code is developed and how it is developed. Consider a software development system shown in Figure 2 which takes code requirements and produces software and other outputs (e.g. documentation).

Verification compares the output of the code with the model and its products (e.g. analytic tests); development of the code is guided by its faithfulness to the model. This is solving the problem (as stated by the model) right.

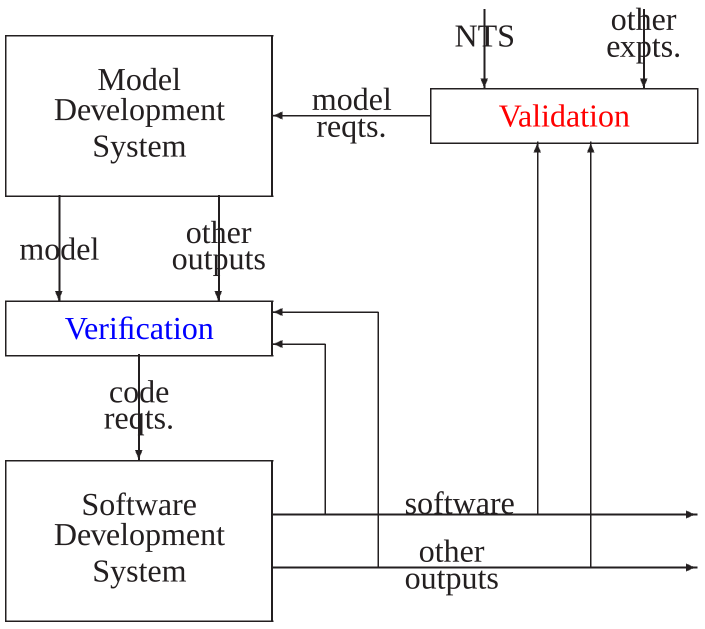

Figure 2: Dynamic Role of Verification and Validation. Verification uses the products of the software development system to guide the software development process. Validation uses the products of the software development system to guide the model development process.

The model that provides the standard for verification is itself the product of a model development system. In this case there is no way to compare the output of the model with its experimental input, since the model is not analytically solvable. Instead we judge the model by using its computational surrogate, which we have verified as equivalent. So validation also defines a standard for code quality, but its purpose is ultimately judging the fidelity of the model.

Validation compares the output of the verified code with experimental data (e.g. NTS); development of the model is guided by its faithfulness to reality. This is solving the right problem (as stated by experiments).

How Do I Verify

My final insight was if verification played an integral role in quality control, then any quality control techniques are candidates for verification. Returning to my original idea of looking at the best practices for verification: how does any industry control the quality of its products?

Lakos provides an answer to this question for cars and for software:

> To summarize: a well-designed car is built from layered parts that have been tested throughly by the manufacturer:
>
> 1. in isolation,
> 2. within a sequence of partially integrated subsystems, and
> 3. as a fully integrated product.
>
> Once assembled, these parts are easily accessible by mechanics to facilitate proper

```
subroutine cg_solve(self, A, b, x)
    ! Conjugate gradient solve with preconditioning.
    type(cg), intent(inout) :: self
    type(crs_matrix), intent(in) :: A
    real, dimension(:), intent(in) :: b
    real, dimension(:), intent(inout) :: x
    REQUIRE(spd(A))
    REQUIRE(size(x) == size(A, 1), size(x) == size(b))
    ! Code...
    ENSURE((self%iteration == self%max_iteration).or.
        (norm(r) <= self%stop_tol*(norm(A)*norm(x)+norm(b))))
```

Figure 3: Contract Specification for a Conjugate Gradient Solver. The preprocessor turns `REQUIRE` and `ENSURE` into executable `if` statements; the subroutine verifies its input and output each time it is called.

> testing and maintenance. In software, the concepts remain essentially the same. –*John Lakos*[Lakos, 1996]

Testing in Isolation. The isolated parts of a software product are its routines: how should one test these parts against the models they implement? Alan Turing considered this question almost 50 years ago:

> How can one check a large routine in the sense that it's right? In order that the man who checks may not have too difficult a task, the programmer should make a number of definite assertions which can be checked individually, and from which the correctness of the whole program easily follows. –*Alan Turing*[Turing, 1950]

Design by Contract™[Meyer, 1995] is a technique for making "definite assertions" a part of the code. It is a specification discipline, an implementation guide, a verification mechanism, and a documentation discipline.

Design by Contract™ explicitly requires:

1. specification of the signature of a routine: given by the explicit type declaration of all argument and return values.

2. specification of the allowable inputs: given in `REQUIRE` statements

3. specification of acceptable output: given in `ENSURE` statements.

for each routine.

Figure 3 shows the contract specification for a conjugate gradient solver. In this case:

1. the type declarations specify the signature of the routine: all of the argument and return types are explicitly declared; this subroutine is in a module so the compiler verifies that each subroutine call has actual argument types that agree with the subroutine declaration.

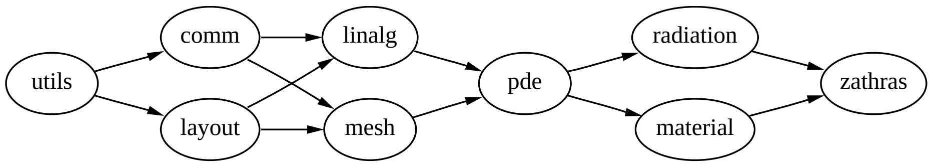

Figure 4: Build Order for Zathras Components. Each component is built in the order shown; unit tests verify the operation of each component before the next component is built.

2. the `REQUIRE` statements specify the allowable inputs: the matrix `A` must be symmetric positive definite and the size of `A` and the arrays `x` and `b` must be conformal

3. the `ENSURE` statement specifies the acceptable output: either the number of iterations reaches the maximum specified or the size of the residual is less than the acceptable error bounds

The `REQUIRE` and `ENSURE` statements are implemented as `m4` macros which expand to standard `Fortran 90 if` statements in the test code, or to comments in the production code. These assertions are turned on throughout the code development and testing phases; each routine verifies its specified operation every time it is called.

Testing in Subsystems. Software Parts (routines) are assembled into subsystems (called packages, modules, clusters, components, or units): how should one test these units? In order to test the functionality of a unit and only the functionality of that unit[2], the code must be designed for testability. Specifically, it should be designed so that it can be built bottom-up in levels as shown in Figure 4.

Starting from the left, each package is built, tested, and approved before the next level is built. Testing in this hierarchical, incremental fashion is called levelized testing; it is an efficient and effective method of assuring code quality:

> Distributed system testing throughout the design hierarchy can be much more effective per testing dollar than testing at only the highest-level interface. –*John Lakos*[Lakos, 1996]

An example of unit testing for partial differential equation (PDE) solvers is the analytic test method (ATM). Given:

- A PDE to be solved: $\mathcal{L}u = f$
- A finite difference solver: $\tilde{u} = L^{-1}f$
- A Remainder term: $R \equiv \mathcal{L} - L$

the ATM tests the finite difference solver by:

1. Solving $R\hat{u} = 0$
2. Finding $\hat{f} = \mathcal{L}\hat{u}$

[2]Testing only the functionality of a unit limits the complexity of the test; the test need be no more (or less) complex than the unit.

Table 1: Marshak Wave Solutions. In addition to the matter dominated solutions (1 and 2) I have added mix (4,5, and 6) and radiation dominated (7 and 8) solutions. Comparing the code to these solutions verifies the fidelity of the integrated model.

Class	Driving Condition	Approx.	Reference
1	$E(0,t) = E_0\delta(t)$	a,c,f	LAUR-82-2625[Pomraning, 1982]
2	$T(0,t) = T_0$	a,c,f	LAMS-2421[Petschek, 1960]
4	$T(0,t) = T_0$	d,f	LA-1709[Barfield, 1954]
5	$E(0,t) = E_0\delta(t)$	c,e	(in production)
6	$T(0,t) = T_0$	c,e	(in production)
7	$E(0,t) = E_0\delta(t)$	b,c	(in production)
8	$T(0,t) = T_0$	b,c	(in production)

a: $C_v \gg 4aT^3$ **c:** $\kappa = \frac{\kappa_0}{T^n}$ **e:** $C_v = C_{v0}T^3$

b: $C_v \ll 4aT^3$ **d:** $\kappa = \kappa_0$ **f:** $C_v = C_{v0}$

3. Solving $\tilde{u} = L^{-1}\hat{f}$

4. Comparing $\tilde{u}$ to $\hat{u}$

ATM forces explicit specification of the PDE being solved and the order of difference scheme being used. These steps can be automated. I have developed Python classes which take a differential operator L and solution $\hat{u}$ and produce the forcing function $\hat{f}$ and boundary conditions suitable for testing.

Testing in Integrated Products. Integrated testing of radiation transport packages consists of comparing analytic solutions to the output of the complete package. One standard set of radiation analytic solutions are the Marshak wave solutions. Marshak[Marshak, 1945] showed that under general conditions if a local source of energy is introduced into a cold purely absorbing medium, and the only mechanism for energy transfer is via radiative processes, then the energy propagates as a thermal wave described by the 1-T radiation-matter energy diffusion equation:

$$(4aT^3 + C_v)\frac{\partial T}{\partial t} = \nabla \cdot \frac{c}{3\sigma}\nabla E \tag{1}$$

The standard Marshak wave solutions, problems 1 and 2, are valid only in the matter dominated regime $C_v \gg 4aT^3$; these solutions ignore the $4aT^3$ term in Equation 1. I have extending the Marshak wave analytic solution series by adding problems 4–6, which are valid in the mix regime; these solutions include all the terms in Equation 1, and problems 7 and 8, which are valid in the radiation dominated regime $C_v \ll 4aT^3$; these solutions ignore the C_v term in Equation 1. Table 1 shows the set of Marshak wave solutions with their driving mechanisms and approximations.

Figure 5 compares the matter dominated solution (problem 2) , Barfield's mixed solution (problem 4), and the radiation dominated solution (problem 8) for a problem in the mixed regime. The actual solution lags behind both approximate solutions because including both $4aT^3$ and C_v terms gives a larger effective heat capacity. The additional Marshak wave solutions permit code verification over a larger range of physical interest.

Analytic tests are an important part of verification, but they cannot be the only verification activity, since, as Dijkstra wisely notes:

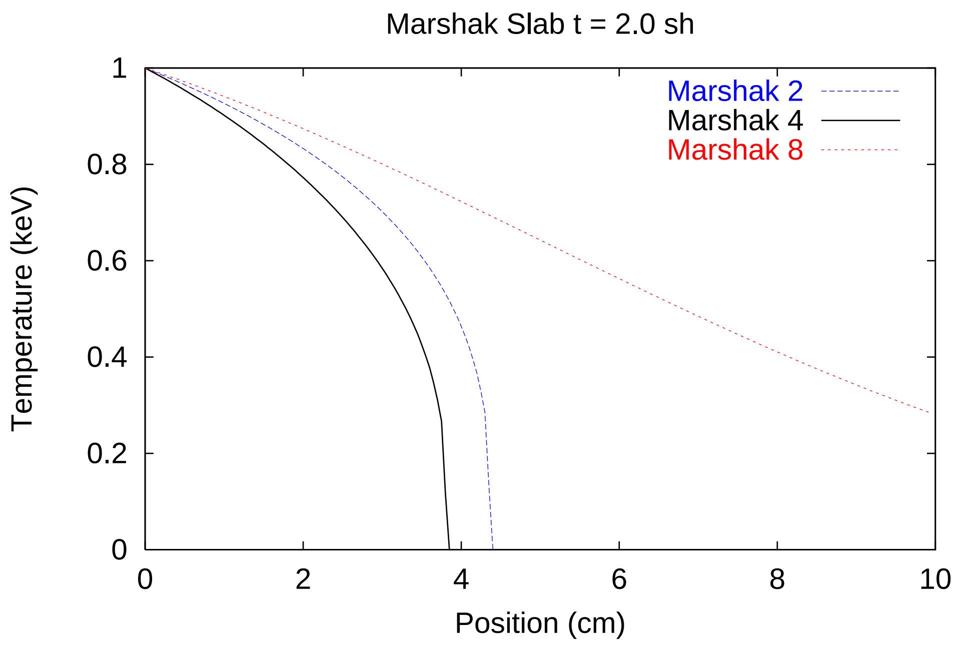

Figure 5: Marshak wave solutions: the matter dominated approximate solution (2), Barfield's exact solution (4) and radiation dominated approximate solution (8) compared for a mixed regime problem. Both approximate solutions lead the exact solution because their effective heat capacities are too small.

> Testing can show the presence of bugs, but not their absence. *–Edsger W. Dijkstra*[Bentley, 1988]

Conclusion

Verification and validation are both necessary for the insight needed for science based stockpile stewardship. Reality, of course, remains the ultimate arbiter, but in the absence of nuclear testing we cannot ask of her the direct questions we could in the past. It now becomes more essential than ever to ensure that our computational models are tied to the mathematical models which are our understanding of reality, and that these models are tied to reality as best we can.

Verification is the assurance that the computational model models the mathematical model. It is the quality control on the code development effort that permeates the analysis, design, implementation, and testing phases. Specific verification activities include:

- Design by Contract™, which makes requirements explicit and testable in a white box view of the system
- Levelized testing, which makes unit testing possible in a grey box view of the system
- Analytic test method, which is a powerful unit testing technique for PDE solvers.
- Analytic solutions, which provide the black box testing of the integrated system

These techniques, applied throughout the software development process, form the basis of the code verification needed for quality ASCI codes.

Appendix

Proof that verification and validation implies model validity.

Theorem 1 *If a code C has been verified against a mathematical model M and validated against a physical system P then M faithfully models P.*

Proof: From Figure 1, verification is the equivalence between the mathematical model's operation, M, and the encoding, E_{cm}, computing, C, and decoding, D_{cm}, process:

Premise 1 (Verification) $D_{mc} \circ C \circ E_{cm} = M$

Similarly, validation is the equivalence between the physical processes operation, P, and the encoding, E_{cp}, computing, C, and decoding, D_{pc}, process:

Premise 2 (Validation) $D_{pc} \circ C \circ E_{cp} = P$

The effect of encoding the physical input into the mathematical model, E_{mp}, solving the mathematical model, M, and decoding the result back into physical terms, D_{pm}, is:

$$\begin{array}{llll} D_{pm}\circ & M & \circ E_{mp} = & \\ D_{pm}\circ & \overbrace{D_{mc}\circ C\circ E_{cm}} & \circ E_{mp} & \text{by Premise 1.} \end{array}$$

We assume that the encoding processes are consistent, specifically:

Axiom 1 $E_{cm} \circ E_{mp} = E_{cp}$;

and similarly that the decoding processes are consistent:

Axiom 2 $D_{pm} \circ D_{mc} = D_{pc}$;

then

$$\begin{array}{cccl} \underbrace{D_{pm}\circ D_{mc}} & \circ C \circ & \underbrace{E_{cm}\circ E_{mp}} = & \\ D_{pc} & \circ C \circ & E_{cp} = & \text{by Axiom 1, 2} \\ & P & & \text{by Premise 2.} \end{array}$$

Q.E.D.

Assert Macros in `m4`. Assertion macros for Design by Contract™ are easy to implement using a preprocessor: here is a minimal implementation for `Fortran 90` using `GNU`'s `m4`:

```
changequote() dnl Reset default m4 quotes...
changequote([,]) dnl then set convenient quotes for f90
changecom([!]) dnl Set comment character for f90

dnl Assertion Macros:

dnl Insist that a single logical argument is true.
```

```
define([INSIST1],
[if (.not.($1)) call Assert_("$1", "__file__", __line__)]) dnl

dnl C assertion model
dnl See: B. W. Kernighan and D. M. Ritchie,
dnl      "The C Programming Language",
dnl      Prentice Hall, NJ
dnl      page 253

dnl If NDEBUG not defined insist that ASSERT's single logical
dnl argument is true.  The order of define and ifdefined is
dnl important: ASSERT must be defined as a test on NDEBUG so
dnl it can be turned on and off at expansion time.

define([ASSERT], [ifdefined([NDEBUG], [], [INSIST1($1)])])dnl
```

Here `Assert_` is a routine which prints an error message such as

`Assertion failed:` *expression*`, file` *filename*`, line` *nnn*

and aborts execution.

More sophisticated assertion models include the assert levels `REQUIRE` and `ENSURE`, multi-line conditions, and non-fatal assertions.

Analytic Test Method Example. The energy deposition split equation for Zathras takes the form:

$$C_v \frac{dT}{dt} = \dot{H} \tag{2}$$

where C_v is the medium's heat capacity, T is the medium's temperature, and $\dot{H}$ is the energy source. This equation is solved using a first order differencing scheme, hence:

$$\begin{aligned} \mathcal{L} &= C_v \frac{d}{dt}, && (3) \\ L &= C_v \frac{\Delta}{\Delta t}, && (4) \\ R \equiv \mathcal{L} - L &= C_v \frac{\Delta t}{2} \frac{d^2}{dt^2}. && (5) \end{aligned}$$

Applying the analytic test method to this system, we:

1. Find the analytic test solution:

$$\begin{aligned} R\hat{T} &= 0, && (6) \\ C_v \frac{\Delta t}{2} \frac{d^2 \hat{T}}{dt^2} &= 0, && (7) \\ \hat{T} &= at + b && (8) \end{aligned}$$

2. Find the analytic forcing function:

$$\hat{\dot{H}} = L\hat{T}, \quad (9)$$

$$\hat{\dot{H}} = C_v \frac{d}{dt}(at+b), \quad (10)$$

$$\hat{\dot{H}} = aC_v. \quad (11)$$

3. Solve the finite difference approximate equation:
   Algebraically:

$$\tilde{T}^{n+1} = \tilde{T}^n + \frac{\Delta t}{C_v} aC_v; \quad \tilde{T}(0) = b \quad (12)$$

   Computationally:

```
T = T + (Delta_t/C_v) * H_dot
```

4. Compare the computational and analytic solutions:
   Algebraically:

$$\tilde{T}^{n+1} = a(n+1)\Delta t + b; \quad (13)$$

   Computationally, if $\hat{T} = 5t + 7$:

t	$\tilde{T}$	$\hat{T}$	$\tilde{T} - \hat{T}$
0.1	7.5	7.5	0.0
0.2	8.0	8.0	0.0
0.3	8.5	8.5	0.0
0.4	9.0	9.0	0.0
0.5	9.5	9.5	0.0
0.6	10.0	10.0	0.0
0.7	10.5	10.5	0.0
0.8	11.0	11.0	0.0
0.9	11.5	11.5	0.0
1.0	12.0	12.0	0.0
1.1	12.5	12.5	0.0
1.2	13.0	13.0	0.0
1.3	13.5	13.5	0.0
1.4	14.0	14.0	0.0
1.5	14.5	14.5	0.0
1.6	15.0	15.0	-1.7763568394e-15
1.7	15.5	15.5	-1.7763568394e-15
1.8	16.0	16.0	0.0
1.9	16.5	16.5	-3.5527136788e-15

Frequently Asked Questions.

1. Is a code-to-code comparison verification or validation?

   That depends on the pedigree of the code and the purpose of the comparison. Consider a new code *C* which is compared to a leagacy code *L*.

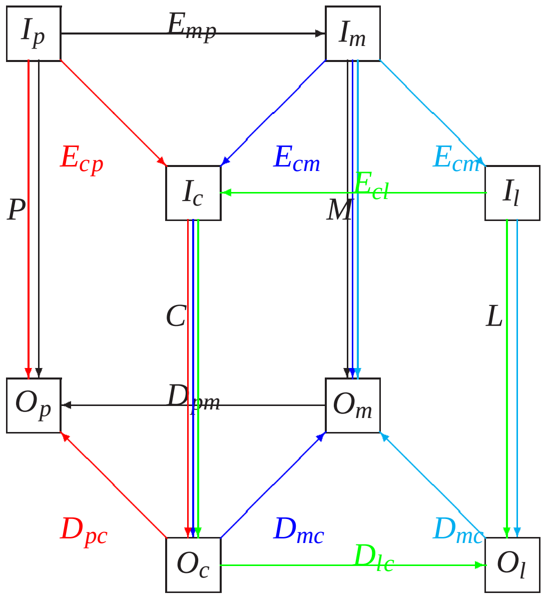

Figure 6: Legacy Code Comparison for Verification. If the legacy code is a surrogate for the model, then comparison to it is verification.

If the legacy code has been verified against a model M, and code C implements that same model, then code L is a surrogate for the model M, and comparing code C's output to code L's output can be used to verify that C correctly implements model M. In this case the code-to-code comparison is verification, as shown in Figure 6.

If the legacy code has been validated against reality P, and especially if code C implements a different model than the legacy code, then code L is a surrogate for reality P, and comparing code C's output to code L's output can be used to validate that C correctly models P. In this case the code-to-code comparison is validation, as shown in Figure 7.

Most code-to-code comparisons are done because the legacy code L has successfully modeled some experiment. In this case the comparison is done for validation purposes.

2. How would you evaluate the quality of the legacy codes?

   The legacy codes are highly successful at doing what was required of them: they are valued by the designers and have been used to design, build, and maintain the stockpile. But if we are to be successful at producing codes with truly predictive capabilities in the absence of testing, then we must hold these new codes to higher standards of verification and validation than in the past.

3. Does the verification and validation diagram imply specific roles for code physicists? Design physicists?

   No. Figure 2 specifies functions that need to be performed, not roles. In the development of my package, I have done model development, verification, and software development. Designers who have contributed to this package have done model development, verification, and validation. Participation in various roles is necessary for the confidence of Code and Design physicists in the final product.

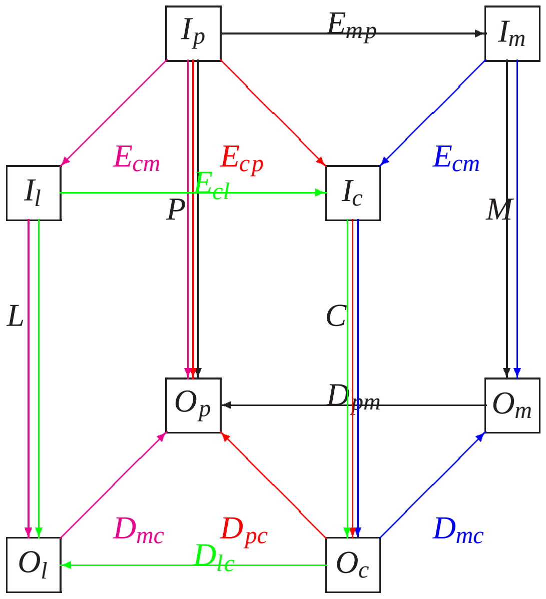

Figure 7: Legacy Code Comparison for Validation. If the legacy code is a surrogate for reality, then comparison to it is validation.

4. What role should a verification and validation group play?

   Designers have often said a code isn't validated until they say its validated. I have argued here that a code isn't verified until the code physicist says it's verified. A verification and validation group can assist the design and code physicist in making these evaluations. In the case of verification a V&V group can provide documentation support for Design by Contract™ , maintain and run unit tests like the ATM, and maintain and run systems test like the Marshak wave series. These support roles can make the code physicist's life **much** easier, but he is still ultimately responsible for the verification of his code package, just as the design physicist is ultimately responsible for the validation of a code.

5. You imply experimental data is a perfect representation of reality. What if it isn't?

   Experimental data is subject to its own methods of validation; the experimentalist is responsible for ensuring that his data is a valid representation of reality. It is no more perfect than the model or code; in all of these cases human judgment is needed to access the quality of the product. The fidelity of experimental data to reality must be taken into account when a model is validated.

6. Isn't SBSS impossible without experiments (nuclear tests)?

   Yes, it would be impossible without nuclear tests. The basic scientific method requires a firm basis in reality through the validation of theory with experiment. However, we do have a wealth of historic nuclear test data which can be used for validation. Whether this historic data, together with non-nuclear experiments, is enough is the crucial ASCI question.

7. Isn't your verification and validation diagram oversimplified? Aren't there many models between (for example) the mathematical and computational models?

One could consider a set of partial differential equations as the mathematical model; the set of corresponding finite difference equations as another model, the (possibly approximate) solution method for the finite difference equations as yet another model, and so on up to the executable code model. My V&V diagram lumps all of these models in one step to illustrate the essence of verification. Verification is any activity which ensures that correspondence of the mathematical model and the computational model, no matter how many models intervene.

8. What is the difference between physics verification and software verification?

   Some people use these terms to distinguish between activities done to ensure faithfulness to the model (e.g. Analytic tests) from those done to ensure software quality (e.g. Design by Contract™, levelized testing). Both of these activities are necessary for verification.

9. What is software quality?

   Quality is value to some person. If ASCI is to provide the understanding necessary for a predictive capability, then it must value the verification and validation activities necessary for insight. A quality ASCI code is one that has (at least) been verified and validated.

10. What one additional thing would you do for verification?

    Formal technical reviews. All of the techniques I've discussed in this paper are designed to convince me that the code is faithful to its model. Formal technical reviews would ensure the community that a code has been verified, without imposing unnecessary bureaucracy.

References

[Hamming, 1986] R. W. Hamming, *Numerical Methods for Scientists and Engineers*, Dover Press (1986)

[Lakos, 1996] J. Lakos, *Large-Scale C++ Software Design*, Addison-Wesley (1996)

[Turing, 1950] A. Turing, *Checking a Large Routine*, talk given at Cambridge University, 24 June 1950

[Meyer, 1995] B. Meyer, *Object Success*, Simon and Schuster (1995)

[Marshak, 1945] R. E. Marshak, Los Alamos Scientific Laboratory Report LA-230, February 1945

[Pomraning, 1982] G. C. Pomraning, *Radiation Hydrodynamics*, LA-UR-82-2625, September 1982

[Petschek, 1960] Albert G. Petschek and Ralph E. Williamson, *The Penetration of Radiation with Constant Driving Temperature*, LAMS-2421, May 1960

[Barfield, 1954] W. D. Barfield, *A Comparison of Diffusion Theory and Transport Theory Results for the Penetration of Radiation into Plane Semi-infinite Slabs*, LA-1709, June 1954

[Bentley, 1988] J. L. Bentley, *More Programming Pearls*, Addison-Wesley 1988